UNIT – 4

Sorting

There are basically two aspects of computer programming. One is data organization also commonly called as data structures. Till now we have seen about data structures and the techniques and algorithms used to access them. The other part of computer programming involves choosing the appropriate algorithm to solve the problem. Data structures and algorithms are linked each other. After developing programming techniques to represent information, it is logical to proceed to manipulate it. This chapter introduces this important aspect of problem solving.

Sorting allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose. For example, a dictionary in which words is arranged in alphabetical order and telephone director in which the subscriber names are listed in alphabetical order. There are many sorting techniques out of which we study the following.

1. Bubble sort
2. Quick sort
3. Selection sort and
4. Heap sort

There are two types of sorting techniques:

1. Internal sorting
2. External sorting

If all the elements to be sorted are present in the main memory then such sorting is called internal sorting on the other hand, if some of the elements to be sorted are kept on the secondary storage, it is called external sorting. Here we study only internal sorting techniques.

7.1. Bubble Sort:

The bubble sort is easy to understand and program. The basic idea of bubble sort is to pass through the file sequentially several times. In each pass, we compare each element in the file with its successor i.e., X[i] with X[i+1] and interchange two element when they are not in proper order. We will illustrate this sorting technique by taking a specific example. Bubble sort is also called as exchange sort.

Example:

Consider the array x[n] which is stored in memory as shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]
33	44	22	11	66	55

Suppose we want our array to be stored in ascending order. Then we pass through the array 5 times as described below:

Pass 1: (first element is compared with all other elements).

We compare X[i] and X[i+1] for i = 0, 1, 2, 3, and 4, and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	Remarks
33	44	22	11	66	55	
	22	44				
		11	44			
			44	66		
				55	66	
33	22	11	44	55	66	

The biggest number 66 is moved to (bubbled up) the right most position in the array.

Pass 2: (second element is compared).

We repeat the same process, but this time we don't include X[5] into our comparisons. i.e., we compare X[i] with X[i+1] for i=0, 1, 2, and 3 and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	Remarks
33	22	11	44	55	
22	33				
	11	33			
		33	44		
			44	55	
22	11	33	44	55	

The second biggest number 55 is moved now to X[4].

Pass 3: (third element is compared).

We repeat the same process, but this time we leave both X[4] and X[5]. By doing this, we move the third biggest number 44 to X[3].

X[0]	X[1]	X[2]	X[3]	Remarks
22	11	33	44	
11	22			
	22	33		
		33	44	
11	22	33	44	

Pass 4: (fourth element is compared).

We repeat the process leaving X[3], X[4], and X[5]. By doing this, we move the fourth biggest number 33 to X[2].

X[0]	X[1]	X[2]	Remarks
11	22	33	
11	22		
	22	33	

Pass 5: (fifth element is compared).

We repeat the process leaving X[2], X[3], X[4], and X[5]. By doing this, we move the fifth biggest number 22 to X[1]. At this time, we will have the smallest number 11 in X[0]. Thus, we see that we can sort the array of size 6 in 5 passes.

For an array of size n, we required (n-1) passes.

7.1.1. Program for Bubble Sort:

```
#include <stdio.h>
#include <conio.h>
void bubblesort(int x[], int n)
{
        int i,  j,  temp;
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n–i-1 ; j++)
                {
                        if (x[j] > x[j+1])
                        {
                                temp = x[j];
                                x[j] = x[j+1];
                                x[j+1] = temp;
                        }
                }
        }
}
```

```
main()
{
        int i, n, x[25];
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter Data:"); for(i = 0; i < n ; i++)
        scanf("%d", &x[i]); bubblesort(x, n);
        printf ("\n Array Elements after sorting: "); for (i = 0; i < n; i++)
        printf ("%5d", x[i]);

}
```

7.2. Selection Sort:

Selection sort will not require no more than n-1 interchanges. Suppose x is an array of size n stored in memory. The selection sort algorithm first selects the smallest element in the array x and place it at array position 0; then it selects the next smallest element in the array x and place it at array position 1. It simply continues this procedure until it places the biggest element in the last position of the array.

The array is passed through (n-1) times and the smallest element is placed in its respective position in the array as detailed below:

Pass 1: Find the location j of the smallest element in the array x [0], x[1], ... x[n-1], and then interchange x[j] with x[0]. Then x[0] is sorted.

Pass 2: Leave the first element and find the location j of the smallest element in the sub-array x[1], x[2], ... x[n-1], and then interchange x[1] with x[j]. Then x[0], x[1] are sorted.

Pass 3: Leave the first two elements and find the location j of the smallest element in the sub-array x[2], x[3], ... x[n-1], and then interchange x[2] with x[j]. Then x[0], x[1], x[2] are sorted.

Pass (n-1): Find the location j of the smaller of the elements x[n-2] and x[n-1], and then interchange x[j] and x[n-2]. Then x[0], x[1], ... x[n-2] are sorted. Of course, during this pass x[n-1] will be the biggest element and so the entire array is sorted.

Time Complexity:

In general we prefer selection sort in case where the insertion sort or the bubble sort requires exclusive swapping. In spite of superiority of the selection sort over bubble sort and the insertion sort (there is significant decrease in run time), its efficiency is also $O(n^2)$ for n data items.

Example:

Let us consider the following example with 9 elements to analyze selection Sort:

1	2	3	4	5	6	7	8	9	Remarks
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap a[i] & a[j]
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap a[i] and a[j]
45	50	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap a[i] and a[j]
45	50	55	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap a[i] and a[j]
45	50	55	60	70	80	75	85	65	Find the fifth smallest element
				i				j	swap a[i] and a[j]
45	50	55	60	65	80	75	85	70	Find the sixth smallest element
					i			j	swap a[i] and a[j]
45	50	55	60	65	70	75	85	80	Find the seventh smallest element
						i j			swap a[i] and a[j]
45	50	55	60	65	70	75	85	80	Find the eighth smallest element
							i	J	swap a[i] and a[j]
45	50	55	60	65	70	75	80	85	The outer loop ends.

7.2.1. Non-recursive Program for selection sort:

```
# include<stdio.h>
# include<conio.h>

void selectionSort( int low, int high );

int a[25];

int main()
{
        int num,  i= 0;
        clrscr();
        printf( "Enter the number of elements: " );
        scanf("%d", &num);
        printf( "\nEnter the elements:\n" );
        for(i=0; i < num; i++)
                scanf( "%d", &a[i] );
        selectionSort( 0, num - 1 );
        printf( "\nThe elements after sorting are: " );
        for( i=0; i< num; i++ )
                printf( "%d    ", a[i] );
```

```
        return 0;
}
void selectionSort( int low, int high )
{
        int i=0, j=0, temp=0, minindex;
        for( i=low; i <= high; i++ )
        {
                minindex = i;
                for( j=i+1; j <= high; j++ )
                {
                        if( a[j] < a[minindex] )
                                minindex = j;
                }
                temp = a[i];
                a[i] = a[minindex];
                a[minindex] = temp;
        }
}
```

7.2.2. Recursive Program for selection sort:

```
#include <stdio.h>
#include<conio.h>
int x[6] = {77, 33, 44, 11, 66};
selectionSort(int);

main()
{
        int i, n = 0;
        clrscr();
        printf (" Array Elements before sorting: ");
        for (i=0; i<5; i++)
                printf ("%d  ", x[i]);
         selectionSort(n);/* call selection sort */
         printf ("\n Array Elements after sorting: ");
         for (i=0; i<5; i++)
                printf ("%d  ", x[i]);
         }
         selectionSort( int n)
         {
                 int k, p,
                 temp, min;
                 if (n== 4)
                    return (-1);
                 min = x[n]; p = n;
                 for (k = n+1; k<5; k++)
                 {
                        if (x[k] <min)
                        {
                                min = x[k]; p
                                = k;
                        }
                 }
                 temp = x[n];          /* interchange x[n] and x[p] */
                 x[n] = x[p];
                 x[p] = temp;
                 n++ ;
}
```

7.3. Quick Sort:

The quick sort was invented by Prof. C. A. R. Hoare in the early 1960's. It was one of the first most efficient sorting algorithms. It is an example of a class of algorithms that work by "divide and conquer" technique.

The quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value. The chosen value is known as the pivot element. Once the array has been rearranged in this way with respect to the pivot, the same partitioning procedure is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers up and down which are moved toward each other in the following fashion:

1. Repeatedly increase the pointer 'up' until a[up] >= pivot.
2. Repeatedly decrease the pointer 'down' until a[down] <= pivot.
3. If down > up, interchange a[down] with a[up]
4. Repeat the steps 1, 2 and 3 till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and place pivot element in 'down' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

1. It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.

2. Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . . . x[high].

3. It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).

4. It calls itself recursively to sort the right sub-array x[j+1], x[j+2], . . x[high] between positions j+1 and high.

The time complexity of quick sort algorithm is of O(n log n).

Algorithm

Sorts the elements a[p], ... ,a[q] which reside in the global array a[n] into ascending order. The a[n + 1] is considered to be defined and must be greater than all elements in a[n]; a[n + 1] = + ∞

```
quicksort (p, q)
{
```

```
      if ( p < q ) then
      {
           call j = PARTITION(a, p, q+1);      // j is the position of the partitioning element
           call quicksort(p, j – 1);
           call quicksort(j + 1 , q);
       }
}

partition(a, m, p)
{
       v = a[m]; up = m; down = p;          // a[m] is the partition element
       do
       {
              repeat
                     up = up + 1;
              until (a[up] ≥ v);

              repeat
                     down = down – 1;
              until (a[down] ≤ v);
              if (up < down) then call interchange(a, up, down);
        } while (up ≥ down);

       a[m] = a[down];
       a[down] = v;
       return (down);
}

interchange(a, up, down)
{
       p = a[up];
       a[up] = a[down];
       a[down] = p;
}
```

Example:

Select first element as the pivot element. Move ‘up’ pointer from left to right in search of an element larger than pivot. Move the ‘down’ pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped.

This process continues till the ‘up’ pointer crosses the ‘down’ pointer. If ‘up’ pointer crosses ‘down’ pointer, the position for pivot is found and interchange pivot and element at ‘down’ position.

Let us consider the following example with 13 elements to analyze quick sort:

1	2	3	4	5	6	7	8	9	10	11	12	13	Remarks
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				up						down			swap up & down
pivot				04						79			
pivot					up			down					swap up & down
pivot					02			57					
pivot						down	up						swap pivot & down
(24	08	16	06	04	02)	38	(56	57	58	79	70	45)	
pivot					down	up							swap pivot & down
(02	08	16	06	04)	24								
pivot, down	up												swap pivot & down
02	(08	16	06	04)									
	pivot	up		down									swap up & down
	pivot	04		16									
	pivot		down	Up									
	(06	04)	08	(16)									swap pivot & down
	pivot	down	up										
	(04)	06											swap pivot & down
	04 pivot, down, up												
				16 pivot, down, up									
(02	04	06	08	16	24)	38							

							(56	57	58	79	70	45)	
							pivot	up				down	swap up & down
							pivot	45				57	
							pivot	down	up				swap pivot & down
							(45)	56	(58	79	70	57)	
							45 pivot, down, up						swap pivot & down
									(58 pivot	79 up	70	57) down	swap up & down
										57		79	
										down	up		
									(57)	58	(70	79)	swap pivot & down
									57 pivot, down, up				
											(70	79)	
											pivot, down	up	swap pivot & down
											70		
												79 pivot, down, up	
							(45	56	57	58	70	79)	
02	04	06	08	16	24	38	45	56	57	58	70	79	

7.3.1. Recursive program for Quick Sort:

```
# include<stdio.h>
# include<conio.h>

void quicksort(int, int);
int partition(int, int);
void interchange(int, int);
int array[25];

int main()
{
        int num, i = 0;
        clrscr();
        printf( "Enter the number of elements: " );
        scanf( "%d", &num);
        printf( "Enter the elements: " );
        for(i=0; i < num; i++)
                scanf( "%d", &array[i] );
        quicksort(0, num -1);
        printf( "\nThe elements after sorting are: " );
```

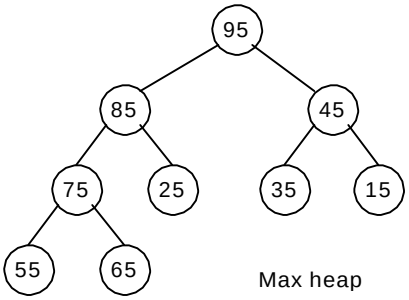
```
        for(i=0; i < num; i++)
                printf("%d  ", array[i]);
        return 0;
}

void quicksort(int low, int high)
{
        int pivotpos;
        if( low < high )
        {
                pivotpos = partition(low, high + 1);
                quicksort(low, pivotpos - 1);
                quicksort(pivotpos + 1, high);
        }
}

int partition(int low, int high)
{
        int pivot = array[low];
        int up = low, down = high;

        do
        {
                do
                        up = up + 1;
                while(array[up] < pivot );

                do
                        down = down - 1;
                while(array[down] > pivot);

                if(up < down)
                        interchange(up, down);

        } while(up < down);
        array[low] = array[down];
        array[down] = pivot;
        return down;
}

void interchange(int i, int j)
{
        int temp;
        temp = array[i];
        array[i] = array[j];
        array[j] = temp;
}
```

7.4. Priority Queue, Heap and Heap Sort:

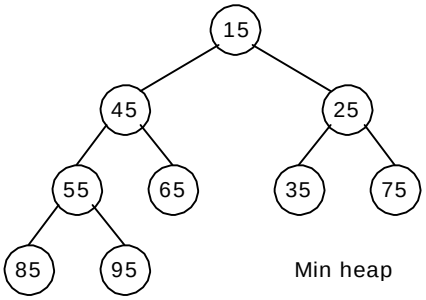
Heap is a data structure, which permits one to insert elements into a set and also to find the largest element efficiently. A data structure, which provides these two operations, is called a priority queue.

7.4.1. Max and Min Heap data structures:

A max heap is an almost complete binary tree such that the value of each node is greater than or equal to those in its children.



Max heap



Min heap

A min heap is an almost complete binary tree such that the value of each node is less than or equal to those in its children.

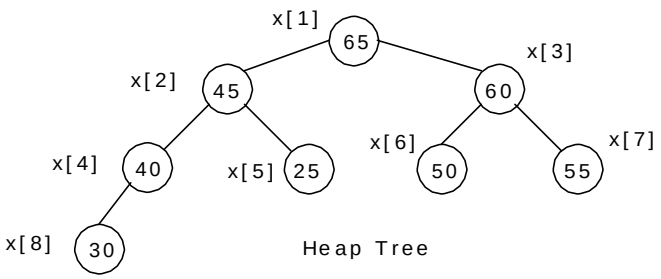
7.4.2. Representation of Heap Tree:

Since heap is a complete binary tree, a heap tree can be efficiently represented using one dimensional array. This provides a very convenient way of figuring out where children belong to.

- The root of the tree is in location 1.
- The left child of an element stored at location i can be found in location 2*i.
- The right child of an element stored at location i can be found in location 2*i+1.
- The parent of an element stored at location i can be found at location floor(i/2).

The elements of the array can be thought of as lying in a tree structure. A heap tree represented using a single array looks as follows:

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
65	45	60	40	25	50	55	30



Heap Tree

7.4.3. Operations on heap tree:

The major operations required to be performed on a heap tree:

1. Insertion,
2. Deletion and
3. Merging.

Insertion into a heap tree:

This operation is used to insert a node into an existing heap tree satisfying the properties of heap tree. Using repeated insertions of data, starting from an empty heap tree, one can build up a heap tree.

Let us consider the heap (max) tree. The principle of insertion is that, first we have to adjoin the data in the complete binary tree. Next, we have to compare it with the data in its parent; if the value is greater than that at parent then interchange the values. This will continue between two nodes on path from the newly inserted node to the root node till we get a parent whose value is greater than its child or we reached the root.

For illustration, 35 is added as the right child of 80. Its value is compared with its parent's value, and to be a max heap, parent's value greater than child's value is satisfied, hence interchange as well as further comparisons are no more required.

As another illustration, let us consider the case of insertion 90 into the resultant heap tree. First, 90 will be added as left child of 40, when 90 is compared with 40 it requires interchange. Next, 90 is compared with 80, another interchange takes place. Now, our process stops here, as 90 is now in root node. The path on which these comparisons and interchanges have taken places are shown by dashed line.

The algorithm Max_heap_insert to insert a data into a max heap tree is as follows:

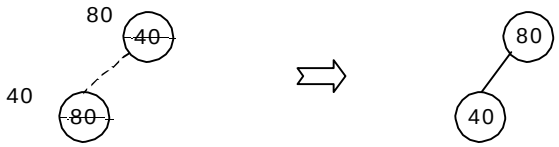
```
Max_heap_insert (a, n)
{
        //inserts the value in a[n] into the heap which is stored at a[1] to a[n-1]
        int i, n;
        i = n;
        item = a[n];
        while ( (i > 1) and (a[ ≤ i/2 ƒ ] < item ) do
        {
                a[i] = a[ ≤ i/2 ƒ ] ;                    // move the parent down
                i = ≤ i/2 ƒ  ;
        }
        a[i] = item ;
        return true ;
}
```

Example:

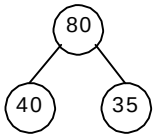
Form a heap using the above algorithm for the data: 40, 80, 35, 90, 45, 50, 70.

1. Insert 40:

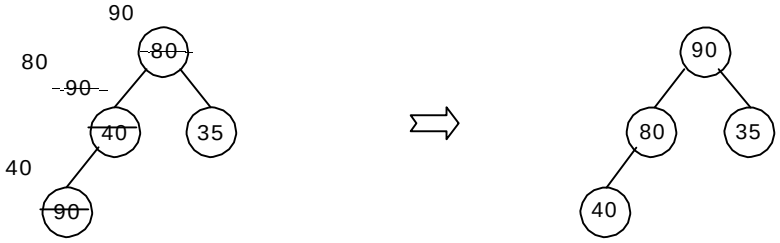
2. Insert 80:



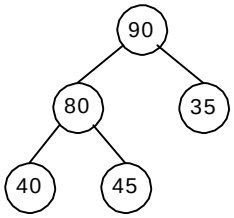
3. Insert 35:



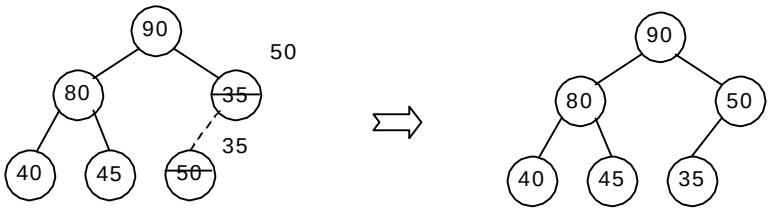
4. Insert 90:



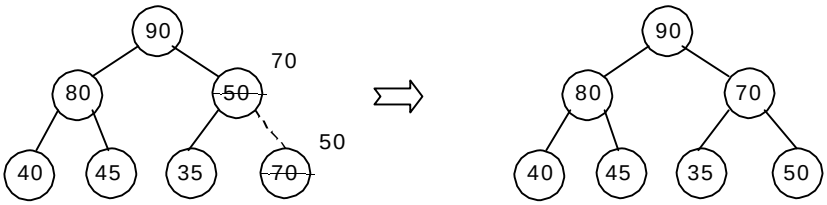
5. Insert 45:



6. Insert 50:



7. Insert 70:



Deletion of a node from heap tree:

Any node can be deleted from a heap tree. But from the application point of view, deleting the root node has some special importance. The principle of deletion is as follows:

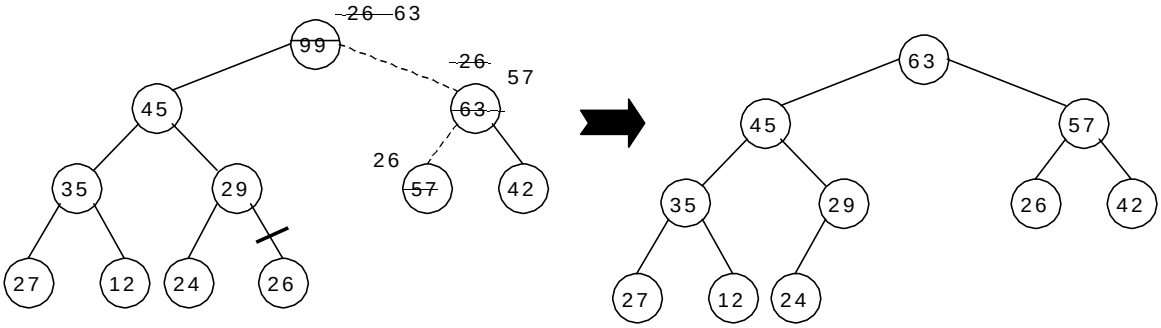
- Read the root node into a temporary storage say, ITEM.
- Replace the root node by the last node in the heap tree. Then re-heap the tree as stated below:
  - Let newly modified root node be the current node. Compare its value with the value of its two child. Let X be the child whose value is the largest. Interchange the value of X with the value of the current node.
  - Make X as the current node.
  - Continue re-heap, if the current node is not an empty node.

The algorithm for the above is as follows:

```
delmax (a, n, x)
// delete the maximum from the heap a[n] and store it in x
{
        if (n = 0) then
        {
                write ("heap is empty");
                return false;
        }
        x = a[1]; a[1] = a[n];
        adjust (a, 1, n-1);
        return true;
}

adjust (a, i, n)
// The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to
form a single heap, 1 ≤ i ≤ n. No node has an address greater than n or less than 1. //
{
        j = 2 *i  ;
        item = a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j ← j + 1;
                        // compare left and right child and let j be the larger child
                if (item ≥ a (j)) then break;
                                                // a position for item is found
                else a[ ≤ j / 2 ƒ ] = a[j]      // move the larger child up a level
                j = 2 * j;
        }
        a [ ≤ j / 2 ƒ ] = item;
}
```

Here the root node is 99. The last node is 26, it is in the level 3. So, 99 is replaced by 26 and this node with data 26 is removed from the tree. Next 26 at root node is compared with its two child 45 and 63. As 63 is greater, they are interchanged. Now,

26 is compared with its children, namely, 57 and 42, as 57 is greater, so they are interchanged. Now, 26 appears as the leave node, hence re-heap is completed.

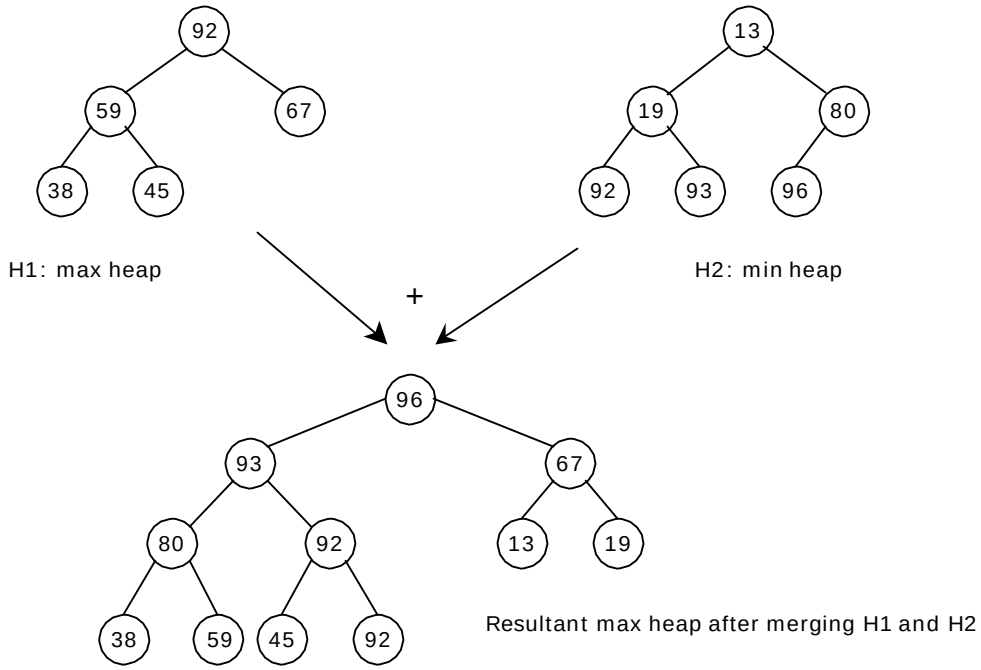


Deleting the node with data 99 After Deletion of node with data 99

7.4.4. Merging two heap trees:

Consider, two heap trees H1 and H2. Merging the tree H2 with H1 means to include all the node from H2 to H1. H2 may be min heap or max heap and the resultant tree will be min heap if H1 is min heap else it will be max heap. Merging operation consists of two steps: Continue steps 1 and 2 while H2 is not empty:

1. Delete the root node, say x, from H2. Re-heap H2.
2. Insert the node x into H1 satisfying the property of H1.



Resultant max heap after merging H1 and H2

7.4.5. Application of heap tree:

They are two main applications of heap trees known are:

1. Sorting (Heap sort) and
2. Priority queue implementation.

7.5. HEAP SORT:

A heap sort algorithm works by first organizing the data to be sorted into a special type of binary tree called a heap. Any kind of data can be sorted either in ascending order or in descending order using heap tree. It does this with the following steps:

1. Build a heap tree with the given set of data.
2. a. Remove the top most item (the largest) and replace it with the last element in the heap.
   b. Re-heapify the complete binary tree.
   c. Place the deleted node in the output.
3. Continue step 2 until the heap tree is empty.

Algorithm:

This algorithm sorts the elements a[n]. Heap sort rearranges them in-place in non-decreasing order. First transform the elements into a heap.

```
heapsort(a, n)
{
        heapify(a, n);
        for i = n to 2 by – 1 do
        {
                temp = a[i];
                a[i] =  a[1];
                a[1] = temp;
                adjust (a, 1, i – 1);
        }
}

heapify (a, n)
//Readjust the elements in a[n] to form a heap.
{
        for i ⇦ ≤ n/2 ƒ to 1 by – 1 do adjust (a, i, n);
}

adjust (a, i, n)
// The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to
form a single heap, 1 ≤ i ≤ n. No node has an address greater than n or less than 1. //
{
        j = 2 *i  ;
        item = a[i] ;
        while (j ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j ⇦ j + 1;
                        // compare left and right child and let j be the larger child
                if (item ≥ a (j)) then break;
                                                // a position for item is found
                else a[ ≤ j / 2 ƒ ] = a[j]      // move the larger child up a level
                j = 2 * j;
        }
        a [ ≤ j / 2 ƒ ] = item;
}
```

Time Complexity:

Each ‘n’ insertion operations takes O(log k), where ‘k’ is the number of elements in the heap at the time. Likewise, each of the ‘n’ remove operations also runs in time O(log k), where ‘k’ is the number of elements in the heap at the time.

Since we always have k ≤ n, each such operation runs in O(log n) time in the worst case.

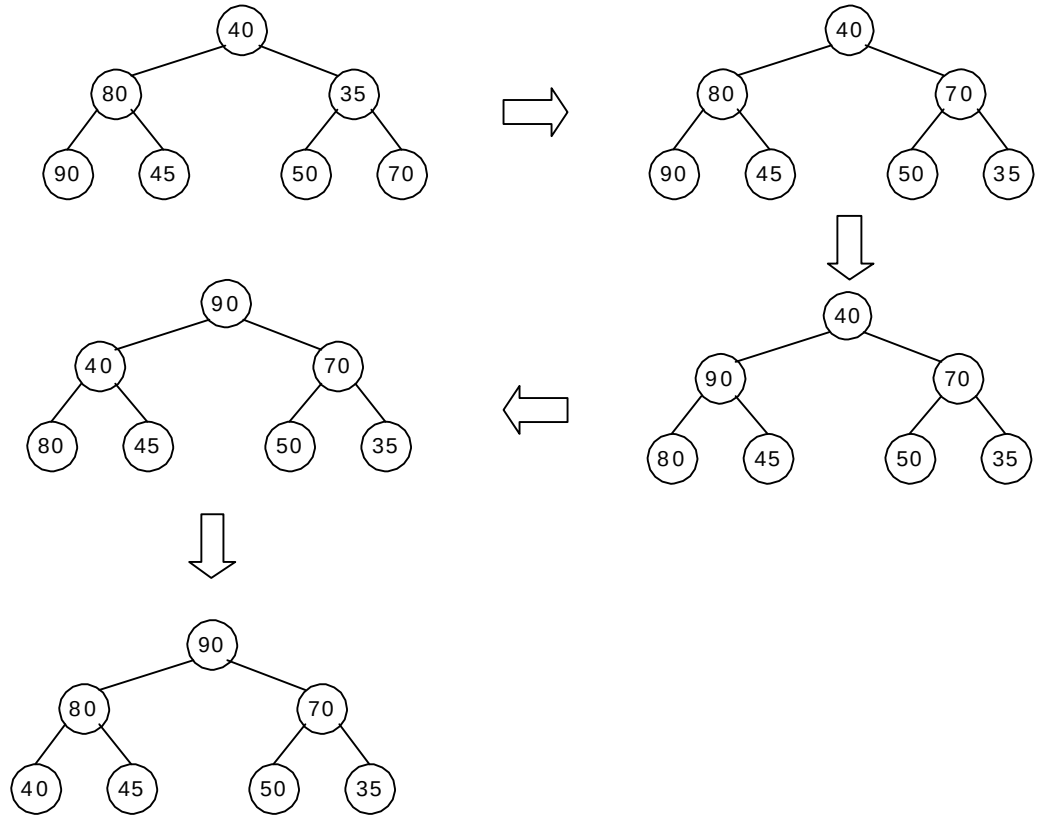
Thus, for ‘n’ elements it takes O(n log n) time, so the priority queue sorting algorithm runs in O(n log n) time when we use a heap to implement the priority queue.

Example 1:

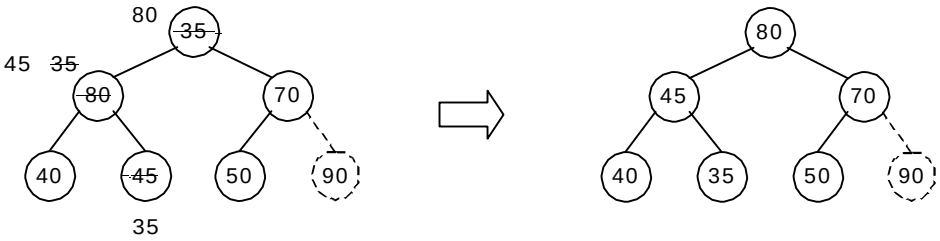
Form a heap from the set of elements (40, 80, 35, 90, 45, 50, 70) and sort the data using heap sort.

Solution:

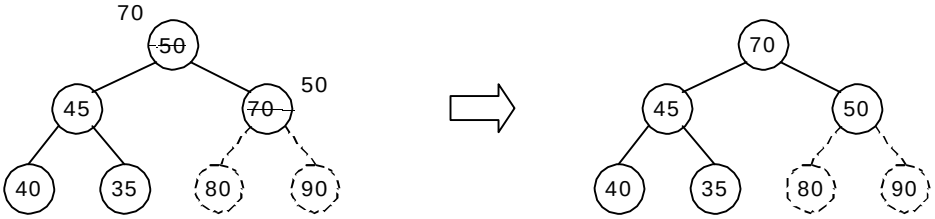
First form a heap tree from the given set of data and then sort by repeated deletion operation:



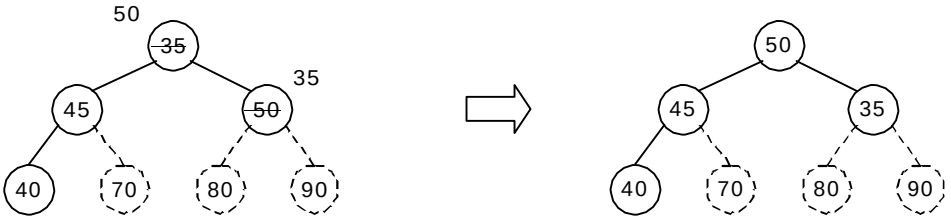
1. Exchange root 90 with the last element 35 of the array and re-heapify



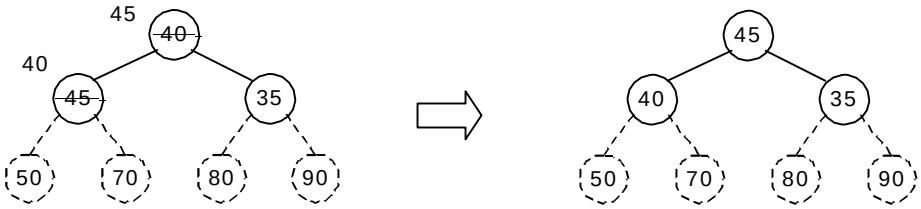
2. Exchange root 80 with the last element 50 of the array and re-heapify



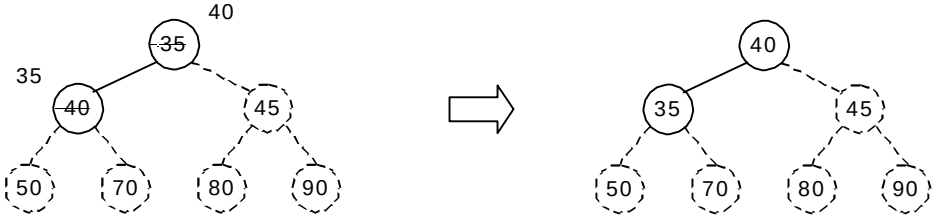
3. Exchange root 70 with the last element 35 of the array and re-heapify



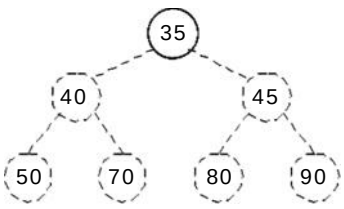
4. Exchange root 50 with the last element 40 of the array and re-heapify



5. Exchange root 45 with the last element 35 of the array and re-heapify



6. Exchange root 40 with the last element 35 of the array and re-heapify



The sorted tree

7.5.1. Program for Heap Sort:

```
void adjust(int i, int n, int a[])
{
	int j, item;
	j = 2 * i;
	item = a[i];
	while(j <= n)
	{
		if((j < n) && (a[j] < a[j+1]))
			j++;
		if(item >= a[j])
			break;
		else
		{
			a[j/2] = a[j];
			j = 2*j;
		}
	}
	a[j/2] = item;
}

void heapify(int n, int a[])
{
	int i;
	for(i = n/2; i > 0; i--)
		adjust(i, n, a);
}

void heapsort(int n,int a[])
{
	int temp, i;
	heapify(n, a);
	for(i = n; i > 0; i--)
	{
		temp = a[i];
		a[i] = a[1];
		a[1] = temp;
		adjust(1, i - 1, a);
	}
}

void main()
{
	int i, n, a[20];
	clrscr();
	printf("\n How many element you want: ");
	scanf("%d", &n);
	printf("Enter %d elements: ", n);
	for (i=1; i<=n; i++)
		scanf("%d", &a[i]);
	heapsort(n, a);
	printf("\n The sorted elements are: \n");
	for (i=1; i<=n; i++)
		printf("%5d", a[i]);
	getch();
}
```

7.6. Priority queue implementation using heap tree:

Priority queue can be implemented using circular array, linked list etc. Another simplified implementation is possible using heap tree; the heap, however, can be represented using an array. This implementation is therefore free from the complexities of circular array and linked list but getting the advantages of simplicities of array.

As heap trees allow the duplicity of data in it. Elements associated with their priority values are to be stored in from of heap tree, which can be formed based on their priority values. The top priority element that has to be processed first is at the root; so it can be deleted and heap can be rebuilt to get the next element to be processed, and so on. As an illustration, consider the following processes with their priorities:

Process	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
Priority	5	4	3	4	5	5	3	2	1	5

These processes enter the system in the order as listed above at time 0, say. Assume that a process having higher priority value will be serviced first. The heap tree can be formed considering the process priority values. The order of servicing the process is successive deletion of roots from the heap.

Exercises

1. Find the expected number of passes, comparisons and exchanges for bubble sort when the number of elements is equal to "10". Compare these results with the actual number of operations when the given sequence is as follows: 7, 1, 3, 4, 10, 9, 8, 6, 5, 2.

2. When a "C" function to sort a matrix row-wise and column-wise. Assume that the matrix is represented by a two dimensional array.

3. A very large array of elements is to be sorted. The program is to be run on a personal computer with limited memory. Which sort would be a better choice: Heap sort or Quick sort? Why?

4. Here is an array of ten integers: 5 3 8 9 1 7 0 2 6 4
   Suppose we partition this array using quicksort's partition function and using 5 for the pivot. Draw the resulting array after the partition finishes.

5. Here is an array which has just been partitioned by the first step of quicksort: 3, 0, 2, 4, 5, 8, 7, 6, 9. Which of these elements could be the pivot? (There may be more than one possibility!)

8. Show the result of inserting 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4, 11, 13, and 2, one at a time, into an initially empty binary heap.

9. Sort the sequence 3, 1, 4, 5, 9, 2, 6, 5 using insertion sort.

10. Show how heap sort processes the input 142, 543, 123, 65, 453, 879, 572, 434, 111, 242, 811, 102.

11. Sort 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 using quick sort with median-of-three partitioning and a cutoff of 3.

Multiple Choice Questions

1. In a selection sort of n elements, how many times is the swap function called in the complete execution of the algorithm? [B]
   A. 1 C. n - 1
   B. n^2 D. n log n

2. Selection sort and quick sort both fall into the same category of sorting algorithms. What is this category? [B]
   A. O(n log n) sorts C. Divide-and-conquer sorts
   B. Interchange sorts D. Average time is quadratic

3. Suppose that a selection sort of 100 items has completed 42 iterations of the main loop. How many items are now guaranteed to be in their final spot (never to be moved again)? [C]
   A. 21 C. 42
   B. 41 D. 43

9. When is insertion sort a good choice for sorting an array? [B]
   A. Each component of the array requires a large amount of memory
   B. The array has only a few items out of place
   C. Each component of the array requires a small amount of memory
   D. The processor speed is fast

10. What is the worst-case time for quick sort to sort an array of n elements? [D]
    A. O(log n) C. O(n log n)
    B. O(n) D. $O(n^2)$

11. Suppose we are sorting an array of eight integers using quick sort, and we have just finished the first partitioning with the array looking like this: [A]
    2 5 1 7 9 12 11 10 Which statement is correct?
    A. The pivot could be either the 7 or the 9.
    B. The pivot is not the 7, but it could be the 9.
    C. The pivot could be the 7, but it is not the 9.
    D. Neither the 7 nor the 9 is the pivot

12. What is the worst-case time for heap sort to sort an array of n elements? [C]
    A. O(log n) C. O(n log n)
    B. O(n) D. $O(n^2)$

13. Suppose we are sorting an array of eight integers using heap sort, and we have just finished one of the reheapifications downward. The array now looks like this: 6 4 5 1 2 7 8 [B]
    How many reheapifications downward have been performed so far?
    A. 1 C. 2
    B. 3 or 4 D. 5 or 6

14. Time complexity of inserting an element to a heap of n elements is of the order of [A]
    A. $\log_2 n$ C. $n \log_2 n$
    B. n^2 D. n

15. A min heap is the tree structure where smallest element is available at the [B]
    A. leaf C. intermediate parent
    B. root D. any where

16. In the quick sort method , a desirable choice for the portioning element will be [C]
A. first element of list
B. last element of list
C. median of list
D. any element of list

17. Quick sort is also known as [D]
A. merge sort
B. bubble sort
C. heap sort
D. none

18. Which design algorithm technique is used for quick sort . [A]
A. Divide and conqueror
B. greedy
C. backtrack
D. dynamic programming

19. Which among the following is fastest sorting technique (for unordered data) [C]
A. Heap sort
B. Selection Sort
C. Quick Sort
D. Bubble sort

20. Running time of Heap sort algorithm is -----. [B]
A. O($\log_2$ n)
B. A. O(n $\log_2$ n)
C. O(n)
D. O(n^2)

21. Running time of Bubble sort algorithm is -----. [D]
A. O($\log_2$ n)
B. A. O(n $\log_2$ n)
C. O(n)
D. O(n^2)

22. Running time of Selection sort algorithm is -----. [D]
A. O($\log_2$ n)
B. A. O(n $\log_2$ n)
C. O(n)
D. O(n^2)

23. The Max heap constructed from the list of numbers 30,10,80,60,15,55 is [C]
A. 60,80,55,30,10,15
B. 80,60,55,30,10,15
C. 80,55,60,15,10,30
D. none

24. The number of swappings needed to sort the numbers 8,22,7,9,31,19,5,13 in ascending order using bubble sort is [D]
A. 11
B. 12
C. 13
D. 14

25. Time complexity of insertion sort algorithm in best case is [C]
A. O($\log_2$ n)
B. A. O(n $\log_2$ n)
C. O(n)
D. O(n^2)

26. Which is a stable sort ? [D]
A. Bubble sort
B. Selection Sort
C. Quick sort
D. none

27. Heap is a good data structure to implement [A]
A. priority Queue
B. Deque
C. linear queue
D. none

28. Always Heap is a [A]
A. complete Binary tree
B. Binary Search Tree
C. Full Binary tree
D. none